CMPSC-265
Data Structures and Algorithms

Zaihan Yang
zyang13@suffolk.edu

Department of Math and Computer Science
Suffolk University

Fall 2019

Notice

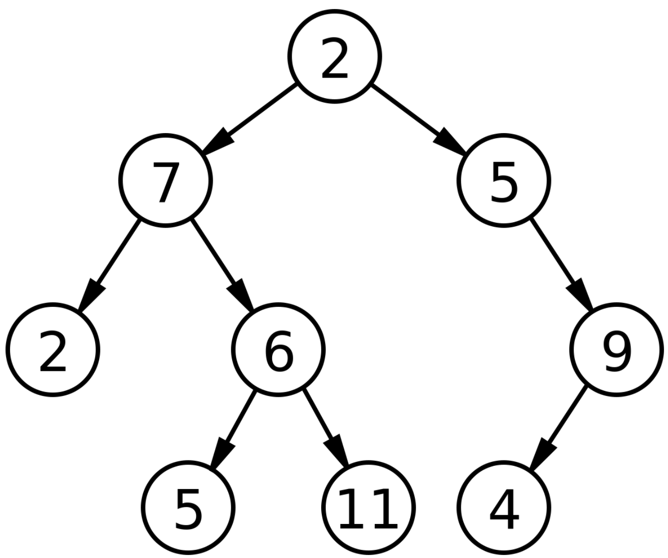
- You will have your Midterm Exam 2 on Nov 25th (Monday)
- Sample Test has been posted on Blackboard.

Review

- Binary Tree
- Binary Search Tree
- Binary Heap & Heap sort
- Hash Table

Binary Tree

- A tree data structure can be defined recursively as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children”).
- A tree is an instance of a graph with no cycles.
- A binary tree is a tree where each node has at most two children.
- Some concepts: root node, leaf node, left child, right child, parent, ancestor, descendant, sibling, leaf node, sub-tree, the size of a binary tree, the depth(height) of a node (and of a tree)
- Basic operations:
  - How to get the size of a binary tree
  - How to find a path on a binary tree
  - How to traverse a binary tree:
    - Level-order traversal
    - Pre-Order traversal
    - In-Order traversal
    - Post-Order traversal
  - ……



Binary Search Tree

- A kind of binary tree and that each node satisfies the following ordering property:
  - Each node’s value is greater than the value of all nodes in its left sub-tree.
  - Each node’s value is less than or equal to the value of all nods in its right sub-tree.
- Basic operations:
  - How to insert a new node into BST
  - How to search for a specific node
  - How to find the node with minimum value
  - How to delete a node from a BST:
    - Three cases
  - Traverse a BST: the same of binary tree
    - In-Order traversal can result in an ascending sorted sequence.

Time Complexity

- The height of a binary tree is within the range of (int)logN ~ N

	Binary Tree			Binary Search Tree		
	Worst	Average	Best	Worst	Average	Best
Search	O(n)	O(h)	O(h)	O(n)	O(h)	O(h)
Insert	O(n)	O(h)	O(h)	O(n)	O(h)	O(h)
Delete	O(n)	O(h)	O(h)	O(n)	O(h)	O(h)

- ** h: is the height of a binary tree (binary search tree)
- For a balanced binary tree, search, insert and delete all costs
- O(logn) in all cases.

Binary Heap

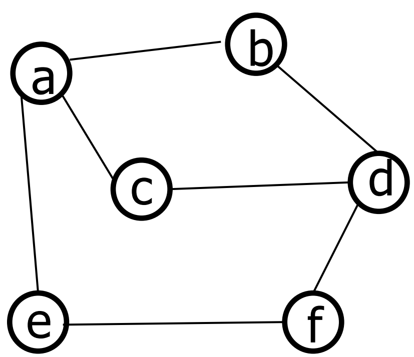
- A binary heap is a complete binary tree satisfies the heap ordering property:
  - Min-heap: each node's value is smaller than or equal to both its two children's.
  - Max-heap: each node's value is greater than or equal to both its two children's.
- A complete binary tree is always balanced.
- Internally, can use an array to represent a binary heap. Suppose the index of a node is at i:
  - Its parent: (i-1)/2; Its left child: i*2 + 1; Its right child: i*2 + 2
- How to reheapify: trickle-down or trickle-up
- Basic operations:
  - How to Insert a new node: first at the end of the array, and possibly trickle-up.
  - How to delete a node: exchange the root node with the last node in the array, and possibly trickle-down.
- Heap Sort: sort an array by repeatedly building the heap.
  - Time complexity: O(nlogn)
  - In-place

Hash table

- A hash table (hash map) is a data structure that can store key-value pairs. A hash table uses a hash function to compute an index, also called a hash value, into an array of buckets or slots, from which the desired value can be found.
- Several things to concern about a hash table:
  - How to choose a good hash function. We often use modular hashing
  - How to represent a non-numeric value to be numeric value, and then compute for the hash value.
  - How to solve the problem of collision:
    - Open addressing:
      - h = (h + s) % tableSize
      - Linea probing: s = 1
      - Quadratic probing: s= 1 ^2, 2^2, 3^2, ….
      - Double hashing: s is determined by another hash function.
    - Separate Chaining

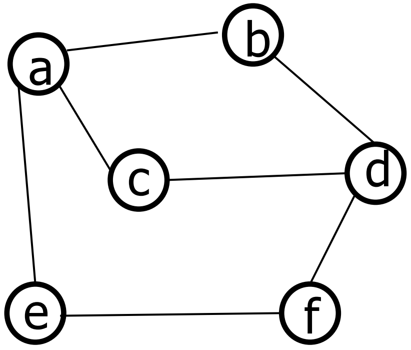
Graphs - Introduction

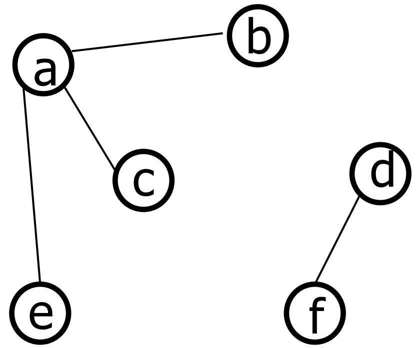
- Graph is a set of vertices (nodes) and edges (links) . Formally, G=(V,E), where V is the vertex set and E is the edge set.
- More general than a tree
  - No root node, cycles (loops) are possible
- Models many real-life scenarios
  - Computer networks, social networks, dependency graph, …



Graphs - Definitions

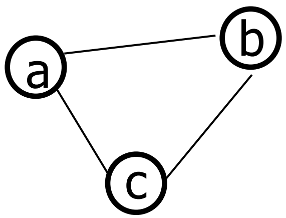
- Two vertices are adjacent to one another (neighbors) if they are connected by a single edge.
- A path is a sequence of edges. There can be more than one path between two vertices.
  - Cycle/loop is a path that starts and ends at the same vertex
- In a Connected Graph there is at least one path between any two vertices.

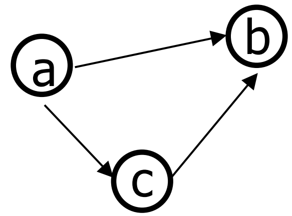




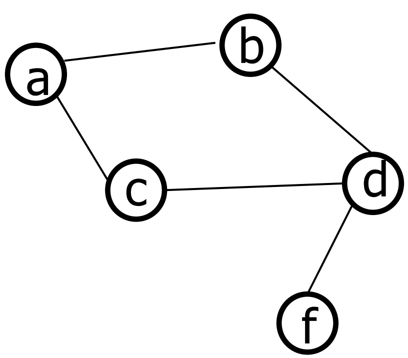
Graphs - Definitions

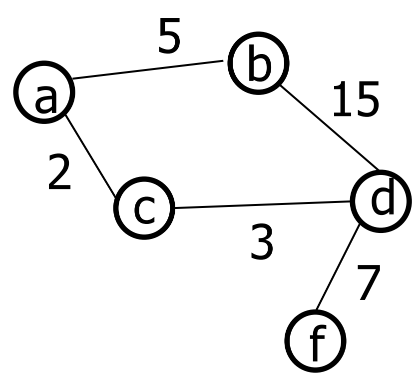
- Directed or non-directed graph
  - Edges have direction or not





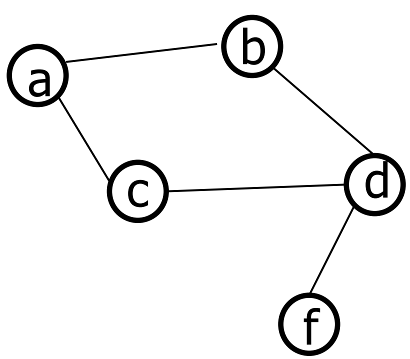
- Weighted or unweighted graph
  - Edges have weights or not





Graph Representation

- Vertices: can simply be labeled/numbered from 0 to N-1
- Edges:
  - Adjacency matrix
  - Adjacency list

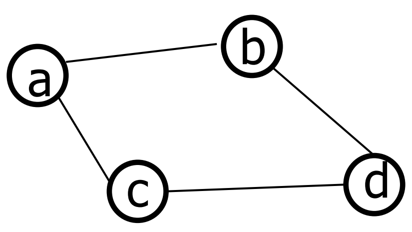



```
class Vertex
{
   public char label; // label (e.g.
'A')
   public boolean wasVisited;

   public Vertex(char lab) //
constructor
   {
     label = lab;
     wasVisited = false;
   }
```

Graph Representation-Adjacency Matrix

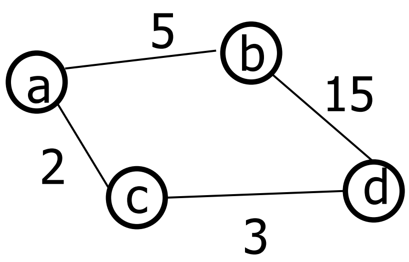
- For a graph with N vertices, the adjacency matrix, A, is a 2-dimensional NxN array.
  - A[i][j]=1 if vertex i and vertex j are adjacent in graph, A[i][j]=0 if i and j are not adjacent.



	a	b	c	d
a	0	1	1	0
b	1	0	0	1
c	1	0	0	1
d	0	1	1	0

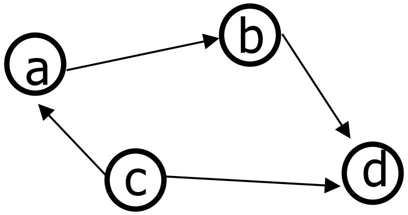
Graph Representation-Adjacency Matrix

- Weighted graph



	a	b	c	d
a	0	5	2	0
b	5	0	0	15
c	2	0	0	3
d	0	15	3	0

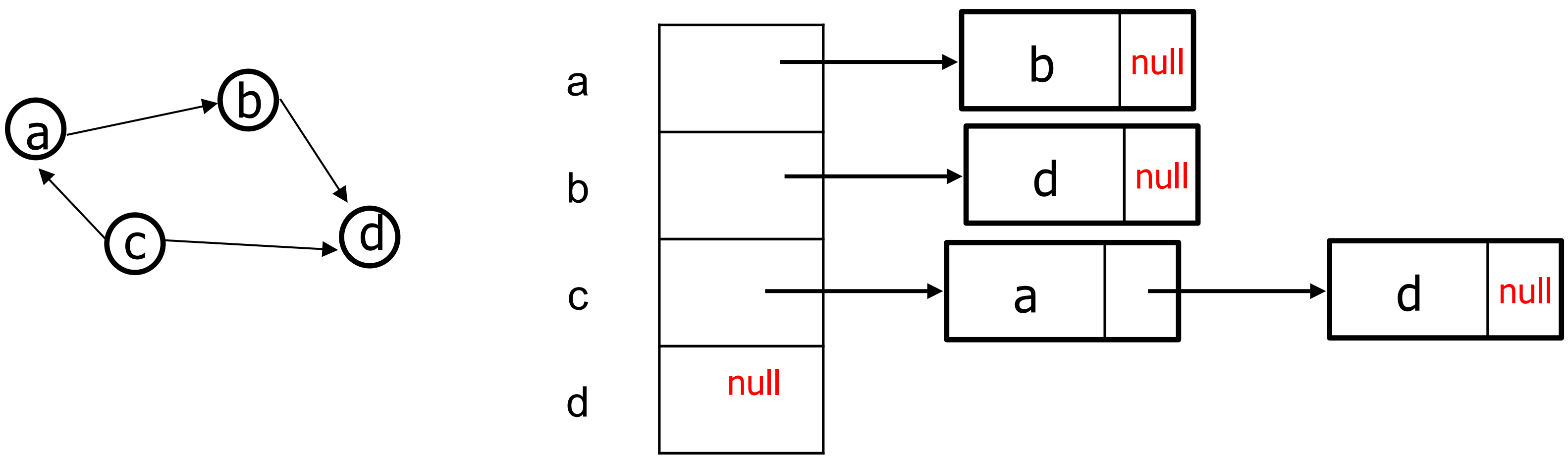
- Directed graph



	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	1	0	0	1
d	0	0	0	0

Graph Representation-Adjacency List

- Array of linked lists
  - It puts the neighbors of a node in a linked list.



A Simple Graph Class

```
public class Graph {
   private int maxSize;
   private Vertex[] nodes; // array of nodes
   private int[][] mat; // adjacency matrix
   private int N;
   public Graph(int maxSize) {
      this.maxSize = maxSize;
      nodes = new Vertex[maxSize];
      mat = new int[maxSize][maxSize];
      for(int i=0; i<maxSize;i++)
         for(int j=0; j<maxSize;j++)
            mat[i][j]=0;
   }
   public void addVertex(char c) {
      if (N >= maxSize) {
        System.out.println("Graph full");
        return;}
      nodes[N++] = new Vertex(c);
   }
   public void addEdge(int v1, int v2) {
      mat[v1][v2] = 1;
      mat[v2][v1] = 1;
   }
}
```

Graph Traversal/Search Methods

- Starting from a vertex, find/visit vertices that can be reached.

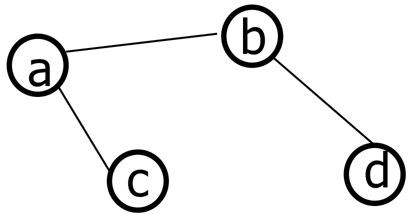
- Two main methods for this purpose:
  - Depth First Search (DFS)
  - Breadth First Search (BFS)

- DFS and BFS work in different orders, but they both visit all connected vertices.

Depth First Search (DFS)

- Go deep as much as you can, Use a stack to remember where to go when you reach a dead end.

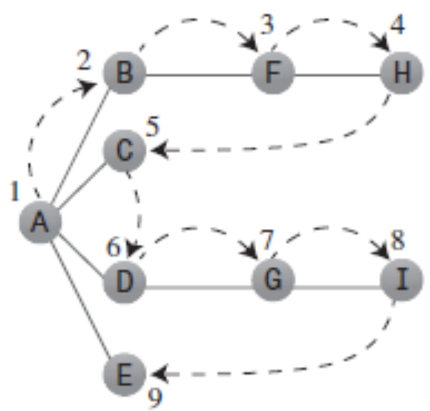
- DFS can be explained by 3 simple rules:
  - Rule 1: If possible, visit an adjacent unvisited vertex, mark it visited, and push it on the stack.
  - Rule 2: If you can't follow rule 1, then if possible pop a vertex from stack.
  - Rule 3: If you can't follow rules 1 and 2, then you are done.



DFS(a): abdc or acbd

DFS-Example

- DFS(A): ABFHCDGIE



Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD
Visit G	ADG
Visit I	ADGI
Pop I	ADG
Pop G	AD
Pop D	A
Visit E	AE
Pop E	A
Pop A	
Done	

Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 625,626

DFS - Implementation

```
public void dfs(){
   nodes[0].visited=true;
   nodes[0].display();
   theStack.push(0);
   while(!theStack.isEmpty()){
      int v=getAdjUnvisitedNode(theStack.peek());
      if (v ==-1) // no such node
         theStack.pop();
      else {
         nodes[v].visited=true;
         nodes[v].display();
         theStack.push(v);
      }
   }
   // stack is empty, reset the flags
   for(int i=0; i<N;i++)
      nodes[i].visited=false;
}
public int getAdjUnvisitedNode(int v){
   for(int i=0; i<N;i++)
      if (mat[v][i]==1 && nodes[i].visited==false)
         return i; // found an unvisited neighbor
   return -1; // no such node
}
```

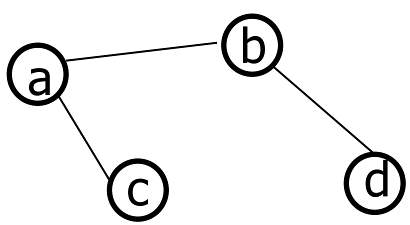
Time complexity? $O(N+|E|)$ where $|E|$ is the number of edges

What happens if we don't mark nodes as visited? infinite loop

Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 633

Breadth First Search (BFS)

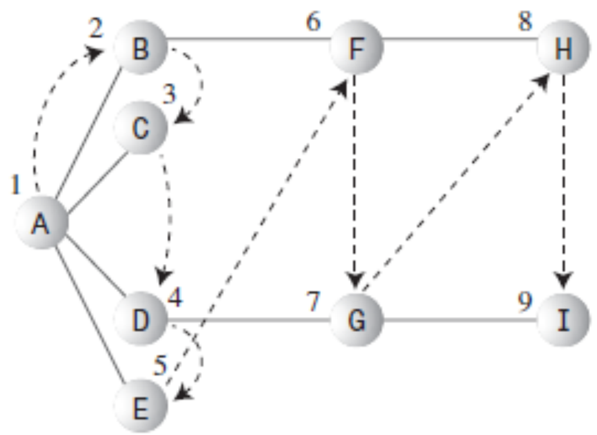
- Unlike DFS which goes as far away as possible from the starting point, BFS likes to stay as close as possible to the starting point. It uses a Queue.
- BFS can be explained by 3 simple rules:
  - Rule 1: visit the next adjacent unvisited vertex, mark it visited, and enqueue it. Go back to current vertex
  - Rule 2: If you can't follow rule 1, then if possible dequeue a vertex from the queue to become your current vertex
  - Rule 3: If you can't follow rules 1 and 2 (queue is empty), then you are done



BFS(a): abcd or acbd

BFS-Example

- BFS(A): ABCDEFGHI



Event	Queue (Front to Rear)
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Remove B	CDE
Visit F	CDEF
Remove C	DEF
Remove D	EF
Visit G	EFG
Remove E	FG
Remove F	G
Visit H	GH
Remove G	H
Visit I	HI
Remove H	I
Remove I	
Done	

Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 6365,637

BFS - Implementation

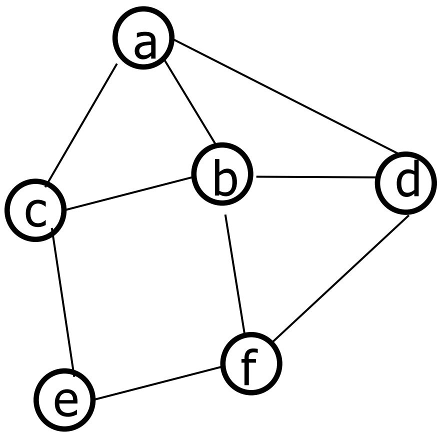
```
public void bfs(){
   nodes[0].visited=true;
   nodes[0].display();
   theQueue.enqueue(0);
   int v2;
   while(!theQueue.isEmpty()){
      int v1=theQueue.dequeue();
      while((v2=getAdjUnvisitedNode(v1))!=-1)
      {
        nodes[v2].visited=true;
        nodes[v2].display();
        theQueue.enqueue(v2);
      }
   }
  // Queue is empty so we can reset the flags
  for(int i=0; i<N;i++)
    nodes[i].visited=false;
}
public int getAdjUnvisitedNode(int v){
   for(int i=0; i<N;i++)
      if (mat[v][i]==1 && nodes[i].visited==false)
         return i; // found an unvisited neighbor
   return -1; // no such node
}
```

Time complexity? O(N+|E|)

Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 641

Example

- BFS and DFS for the following graph starting from a



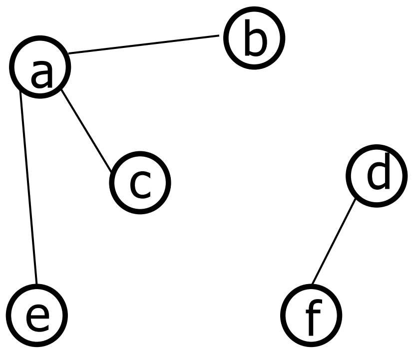
Here is one possible result:
BFS: acbdef
DFS: acefdb

More on DFS/BFS

- Using DFS/BFS, we can detect properties on the Graph:
  - Find all connected component on the graph
  - Given any two nodes, determine whether there is a path between them
  - Given a graph, determine whether it is a connected graph.
  - Given a graph, determine whether it contains a cycle.

More on DFS/BFS

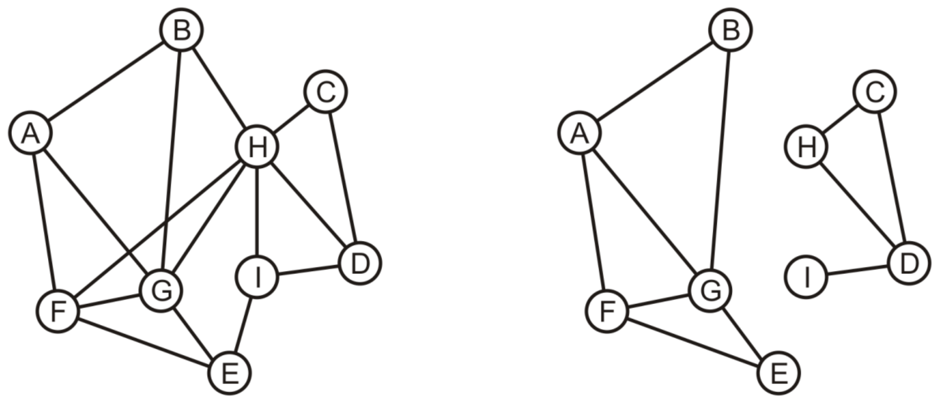
- how do we do DFS/BFS on a graph with multiple components (not a connected graph)?



- If you cannot continue, start DFS/BFS from another unvisited node, until all graph nodes are visited.
  - Need to loop through unvisited nodes

Checking connectedness

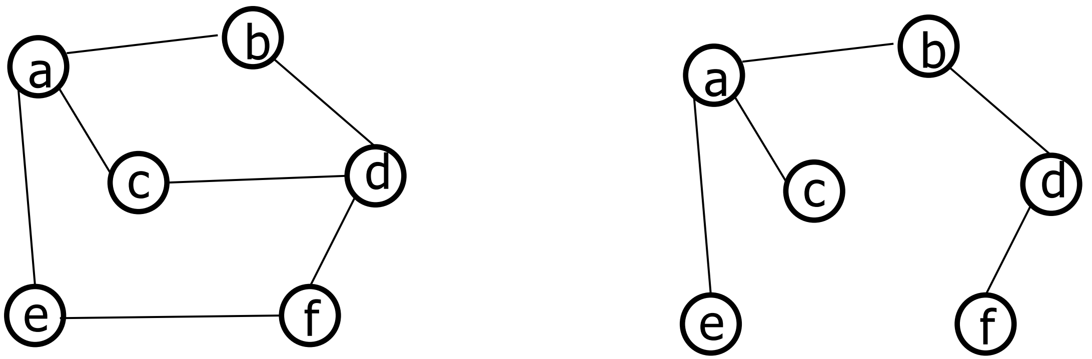
- Given two nodes, how do we check if there is a path between them?



  - Start DFS/BFS on one of the nodes, return true if you see the other one as you go. If the search is done, return false.
- Given a graph, how do we check if it is a connected graph?
  - Start DFS/BFS from any unvisited node, see if you can visit all nodes.

Detecting a cycle

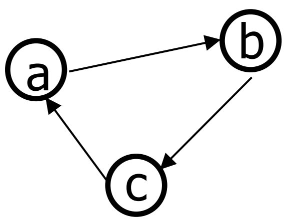
- Given a graph, how do we check if there is a cycle?

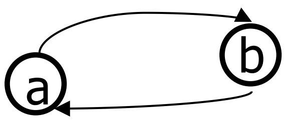


  - Start DFS, return true if you see a visited node which is not your parent (calling node). If the search is done, return false.
  - Can we do BFS as well?

Detecting a cycle in directed graphs

- Given a directed graph, how do we check if there is a cycle?

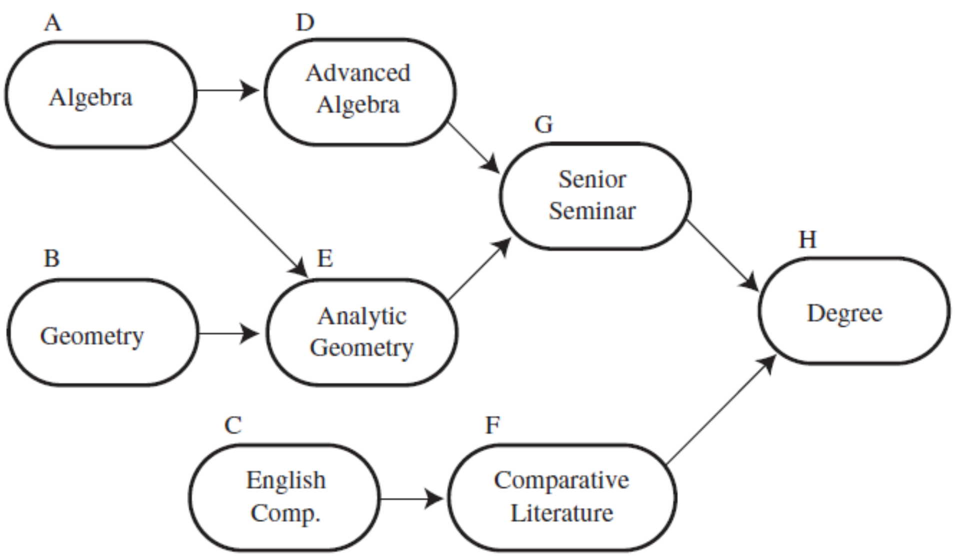




- The previous approach does not work.
- Start DFS, consider 3 sets (visited, unvisited, under visit). As you go mark nodes as under visit, if a node has all neighbors visited, mark it as visited.
  - return true if you see an under visit node.

Topological Sorting in Directed Graphs

- Nodes must be traversed in a specific order.
- Example: course prerequisites
  - ABCDEFGH is one possible way to get a degree, satisfying prerequisites.



Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 650

Topological Sorting in Directed Graphs

- Three steps:
  - Find a vertex that has no successors (the successors to a vertex are those vertices that are directly downstream from it connected by a directed edge).
  - Delete this vertex from the graph, and insert its label at the beginning of the list (insertFirst() in LinkedList).
  - Repeat steps 1 and 2 until all vertices are gone. At this point, the list shows the vertices arranged in topological order.

- What if there is a cycle?
  - It does not work. It works on Directed Acyclic Graphs (DAG).

Topological Sorting in Directed Graphs

```
public void topo() // topological sort
{
  int orig_nVerts = nVerts; // remember how many verts
  while(nVerts > 0) // while vertices remain,
  {
   // get a vertex with no successors, or -1
   int currentVertex = noSuccessors();
   if(currentVertex == -1) // must be a cycle
    {
      System.out.println("ERROR: Graph has cycles");
      return;
     }
   // insert vertex label in topo-sorted array (start at end)
   sortedArray[nVerts-1] = vertexList[currentVertex].label;
   deleteVertex(currentVertex); // delete vertex
  } // end while

 // vertices all gone; display sortedArray
 System.out.print("Topologically sorted order: ");
 for(int j=0; j<orig_nVerts; j++)
    System.out.print( sortedArray[j] );
 System.out.println("");
} // end topo
```

How to find a vertex with no successors?

- Adjacency Matrix: A row with all zeros
- Adjacency List: array cell with an empty list (null entry)

How to delete a vertex?

Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 654

Topological Sorting using DFS

- We can use a similar idea as DFS to do topological sorting:
  - Start from any node, mark it as visited
  - Instead of printing the current node first(as in DFS), recursively call topological sorting on all its unvisited neighbors.
  - When the current node is fully explored (no more unvisited neighbors left), push it to a stack.
  - Continue until no more unvisited nodes left.

- At the end, stack contains the topological sort.